



中山大学 软件工程学院
SUN YAT-SEN UNIVERSITY SCHOOL OF SOFTWARE ENGINEERING

《Rust语言与内存安全设计》

第15讲 并发 + 模式匹配

课程负责人：陈文清 助理教授
chenwq95@mail.sysu.edu.cn

2023年12月06日

上节课：并发



- 1 无畏并发 (Fearless Concurrency)
- 2 多线程
- 3 使用消息跨线程传递数据
- 4 共享状态的并发
- 5 使用Sync和Send来扩展并发



所有权问题：多线程之间如何共享某个值的所有权？

使用共享来实现并发



- 我们希望有一个剩余票数计数器，在所有线程之间共享
-但以“安全”的方式共享（没有数据竞争）
- **Rust** 允许使用引用计数实现共享所有权
 - 获取你想要共享的东西，并将其与引用计数一起分配到堆上
 - 每当您与其他所有者共享对象时，请递增引用计数



1

使用共享来实现并发



- 我们希望有一个在所有线程之间共享的 `remaining_tickets` 计数器
- **Rust** 允许使用引用计数实现共享所有权
 - 将你想要共享的东西和引用计数一起分配到堆上
 - 每当您与其他所有者共享对象时，请递增引用计数
 - 当所有者丢弃对象时，递减引用计数



2





我们已经学过智能指针Rc<T>和RefCell<T>

如果和多线程结合的时候，应该怎么办？

防止多线程情况下的数据竞争？

使用共享来实现并发



■ 多线程的**多重所有权**

- 结合引用计数
- 使用原子引用计数类型（atomically reference counted type），Arc<T>。
- 为什么所有的基础类型都不是原子的，为什么标准库类型不默认使用Arc<T>？
 - 需要性能作为代价
- Arc<T>和Rc<T>的API是相同的。

使用共享资源



```
use std::sync::{Mutex, Arc};
```

```
use std::thread;
```

► Run | Debug

```
fn main() {
```

```
    let counter: Arc<Mutex<i32>> = Arc::new(data: Mutex::new(0));
```

```
    let mut handles: Vec<JoinHandle<()>> = vec![];
```

```
    for _ in 0..10 {
```

```
        let counter: Arc<Mutex<i32>> = Arc::clone(self: &counter);
```

```
        let handle: JoinHandle<()> = thread::spawn(move || {
```

```
            let mut num: MutexGuard<i32> = counter.lock().unwrap();
```

```
            *num += 1;
```

```
        });
```

```
        handles.push(handle);
```

```
    }
```

```
    for handle: JoinHandle<()> in handles {
```

```
        handle.join().unwrap();
```

```
    }
```

```
    println!("Result: {}", *counter.lock().unwrap());
```

```
}
```

多线程的多重锁

- 结合引用计数
- 使用原子引用

使用共享来实现并发



■ RefCell<T>/Rc<T> vs. Mutex<T>/Arc<T>

- Cell家族提供内部可变性，Mutex<T>也提供
- 使用RefCell<T>来改变Rc<T>里面的内容
- 使用Mutex<T>来改变Arc<T>里面的内容
- 注意：Mutex<T>有死锁风险

Send和Sync Trait



■ Rust语言的并发特性较少，目前讲的都是来自标准库（而不是语言本身）

- 无需局限于标准库的并发，可以自己实现并发
- 但在Rust语言中有两个并发概念：
 - `std::marker::Sync`和`std::marker::Send`两个trait

Send和Sync Trait



■ Send: 允许线程间转移所有权

- 实现Send trait的类型可在线程间转移所有权
- Rust中几乎所有的类型都实现了Send:
 - 但Rc<T>没有实现Send，它只用于单线程情景
- 任何完全由Send类型组成的类型也被标记为Send
- 除了原始指针之外，几乎所有的基础类型都是Send

```
// 通过实现 Send trait, 类型可以安全地在线程之间传递  
struct MyType;  
  
unsafe impl Send for MyType {}
```

Send和Sync Trait



■ Sync: 允许从多线程访问

- 实现Sync的类型可以安全的被多个线程引用
- 也就是说：如果T是Sync，那么&T就是Send
 - 引用可以被安全的送往另一个线程
- 基础类型都是Sync
- 完全由Sync类型组成的类型也是Sync
 - 但，Rc<T>不是Sync的
 - RefCell<T>和Cell<T>家族也不是Sync的
 - 而Mutex<T>是Sync的

```
// 通过实现 Sync trait, 类型可以在多个线程中安全地被引用
1 implementation
struct MyType;

unsafe impl Sync for MyType {}
```

Send和Sync Trait



■ 手动实现Send和Sync是不安全的

- 因为由 Send 和 Sync 特征组成的类型也会自动是Send和 Sync的，所以我们不必手动实现这些特征。
- 作为标记特征（Marker Trait），它们甚至没有任何方法需要实现。它们只是用于强制执行与并发相关的不变量。
- 手动实现这些特征涉及实现不安全的 Rust 代码。

Send和Sync Trait



■ 手动实现Send和Sync是不安全的

- Send和Sync trait也是自动实现的
- 但需要注意的是，如果类型中包含了Cell、RefCell等内部可变性（interior mutability）的结构，需要手动声明unsafe impl Sync。

```
// 包含内部可变性的类型
#[derive(Debug)]
struct SharedData {
    data: RefCell<Vec<i32>>,
}

// 实现 Send 和 Sync trait
unsafe impl Send for SharedData {}
unsafe impl Sync for SharedData {}
```

Send和Sync Trait

■ 手动实现Send和Sync是不安全的

- Send和Sync trait也是自动实现的
- 但需要注意的是，如果类型中包含了Cell要手动声明unsafe impl Sync。

```
// 包含内部可变性的类型
#[derive(Debug)]
struct SharedData {
    data: RefCell<Vec<i32>>,
}

// 实现 Send 和 Sync trait
unsafe impl Send for SharedData {}
unsafe impl Sync for SharedData {}
```

```
// 在多个线程中共享 SharedData 实例
let shared_data = Arc::new(SharedData {
    data: RefCell::new(vec![1, 2, 3]),
});

let (tx, rx) = mpsc::channel();

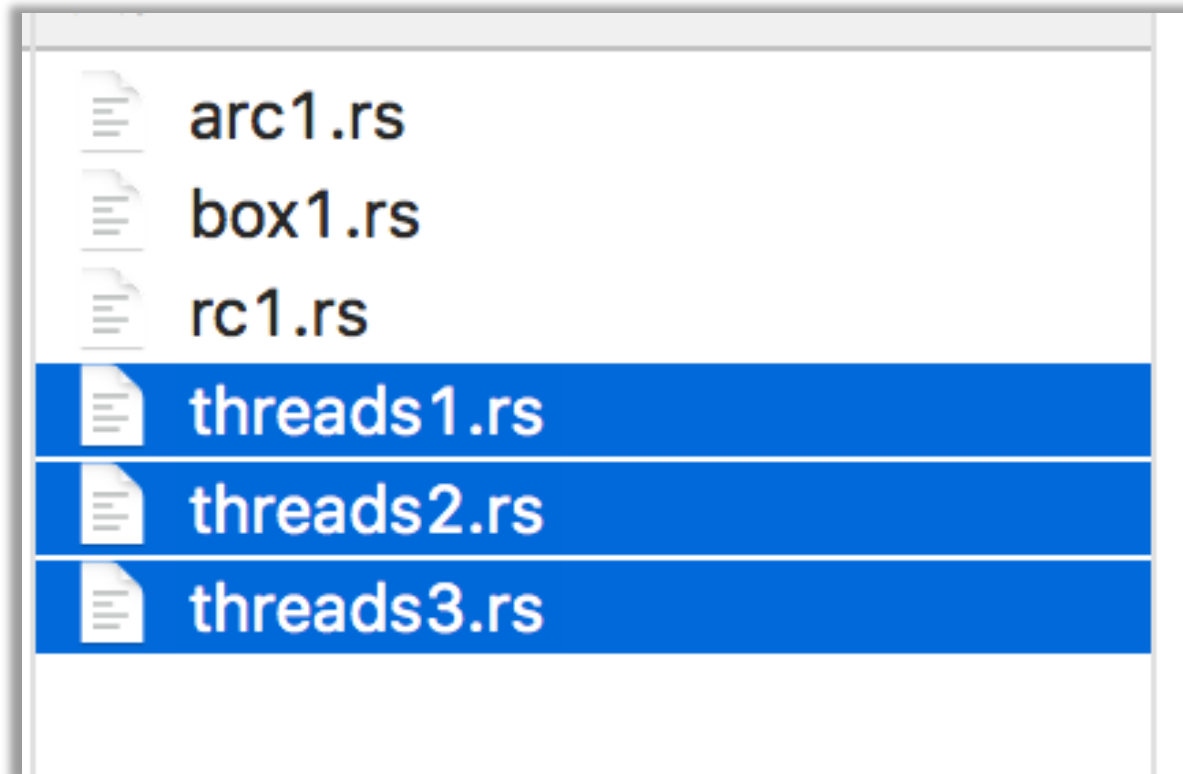
let thread1 = {
    std::thread::spawn(move || {
        {
            let mut data = shared_data.data.borrow_mut();
            data.push(4);
        }
        // 可变引用在这里被释放
        tx.send(shared_data).unwrap();
    })
};

let thread2 = {
    std::thread::spawn(move || {
        let received = rx.recv().unwrap();
        println!("{:?}", *received);
    })
};

thread1.join().unwrap();
thread2.join().unwrap();
```

- 1 无畏并发 (Fearless Concurrency)
- 2 多线程
- 3 使用消息跨线程传递数据
- 4 共享状态的并发
- 5 使用Sync和Send来扩展并发

课堂练习





本节课另一部分知识点：模式和匹配

类型系统（高级用法）



■ 语句（statements）与表达式（expressions）

- 语句是执行某些操作且不返回值的指令。表达式是去计算并返回一个结果值。
- 表达式可以是语句的一部分：

```
fn main() {  
    let y = {  
        let x = 3;  
        x + 1  
    };  
    println!("The value of y is: {y}");  
}
```

Diagram illustrating the distinction between statements and expressions in Rust code:

- The entire block `let y = { ... };` is enclosed in a red box and labeled **语句** (Statement).
- Inside this block, the line `let x = 3;` is labeled **语句** (Statement).
- The expression `x + 1` is labeled **表达式** (Expression).

■ let语句

- 基本用法：创建新的变量绑定（如`let x = 5;`）；
- 本质上，**let**语句是一个模式匹配语句：
 - 在“=”左侧提供标识符
 - 在代数数据类型的字段中提取值

类型系统（高级用法）



■ let语句

- 基本用法：创建新的变量绑定（如`let x = 5;`）；
- 本质上，`let`语句是一个模式匹配语句：
 - 在“=”左侧提供标识符

```
struct Items(u32);  
▶ Run | Debug  
fn main() {  
    let items: Items = Items(2);  
    let mut another_item: Items = Items(2);  
  
    let items_ptr: &Items = &items;  
    let ref items_ref: &Items = items;  
}
```

类型系统（高级用法）



■ let语句

➤ shadowing: 同名覆盖

```
fn main() {  
    let x = 5;  
  
    let x = x + 1;  
  
    {  
        let x = x * 2;  
        println!("The value of x in the inner scope is: {x}");  
    }  
  
    println!("The value of x is: {x}");  
}
```

类型系统（高级用法）



■ let语句

➤ shadowing: 同名覆盖

The value of x in the inner scope is: 12
The value of x is: 6

```
fn main() {  
    let x = 5;  
  
    let x = x + 1;  
  
    {  
        let x = x * 2;  
        println!("The value of x in the inner scope is: {x}");  
    }  
  
    println!("The value of x is: {x}");  
}
```

类型系统（高级用法）



■ let语句

- 本质上，let语句是一个模式匹配语句：
- 另一用法：在代数数据类型的字段中提取值

类型系统（高级用法）



■ let语句

➤ 本质上，let语句是一个模式匹配语句：

➤ 在代码

```
struct Order {  
    count: u8,  
    item: Food,  
    payment: PaymentMode  
}  
  
fn main() {  
    let food_order = Order { count: 2,  
                             item: Food::Salad,  
                             payment: PaymentMode::Credit };  
  
    // let can pattern match inner fields into new variables  
    let Order { count, item, .. } = food_order;  
}
```

类型系统（高级用法）



■ 类型转换

➤ Rust不会自动执行类型转换：

```
fn main() {  
    let foo: u32 = 5;  
    let bar: i32 = 6;  
    let difference = foo - bar;  
    println!("{}", difference);  
}
```

类型系统（高级用法）



■ 类型转换

➤ Rust不会自动执行类型转换：

```
fn main() {  
    let foo: u32 = 5;  
    let bar: i32 = 6;  
    let difference = foo - bar;  
    println!("{}", difference);  
}
```

error[E0277]: cannot subtract `i32` from `u32`

--> src/main.rs:6:26

6

let difference = foo - bar;

^ no implementation for `u32 - i32`

类型系统（高级用法）



■ 类型转换

➤ Rust不会自动执行类型转换：

```
fn main() {  
    let foo: u32 = 5;  
    let bar: i32 = 6;  
    let difference = foo - bar;  
    println!("{}", difference);  
}
```

error[E0277]: cannot subtract `i32` from `u32`

--> src/main.rs:6:26

```
6 |         let difference = foo - bar;  
    ^ no implementation for `u32 - i32`
```

类型系统（高级用法）



■ 类型转换

- 使用as关键词进行强制类型转换：

▶ Run | Debug

```
fn main() {  
    let a: u8 = 34u8;  
    let b: u64 = a as u64;  
    let c: f32 = b as f32;  
}
```

类型系统（高级用法）



■ 类型转换

- 使用as关键词进行强制类型转换：

```
use std::fmt::Display;
▶ Run | Debug
fn main() {
    let a: String = "hello".to_string();
    let b: &String = &a;
    let c: &dyn Display = b as &dyn Display;
}
```

“引用”转换为“特征对象”

类型系统（高级用法）



■ 类型别名 (alias)

- 并非Rust独有的特性，C语言有typedef关键词
- 使代码更具有可读性

```
type MyInt = i32;  
type MyString = String;
```

类型系统（高级用法）



■ 类型别名 (alias)

- 并非Rust独有的特性，C语言有typedef关键词
- 使代码更具有可读性

```
0 implementations
pub struct ParsedPayload<T> {
|   inner: T
|
| }
0 implementations
pub struct ParseError<E> {
|   inner: E
|
| }
type ParserResult<T, E> = Result<ParsedPayload<T>, ParseError<E>>;
pub fn parse_payload<T, E>(stream: &[u8]) -> ParserResult<T, E> {
|   unimplemented!();
|
| }
```


if let 条件表达式



- 等同于只关心一个情况的 match 语句简写

```
fn main() {  
    let favorite_color: Option<&str> = None;  
    let is_tuesday = false;  
    let age: Result<u8, _> = "34".parse();  
  
    → if let Some(color) = favorite_color {  
        println!("Using your favorite color, {}, as the background", color);  
    → } else if is_tuesday {  
        println!("Tuesday is green day!");  
    → } else if let Ok(age) = age {  
        if age > 30 {  
            println!("Using purple as the background color");  
        } else {  
            println!("Using orange as the background color");  
        }  
    } else {  
        println!("Using blue as the background color");  
    }  
}
```

while let 条件表达式



- 允许只要模式匹配就一直进行 while 循环

```
let mut stack = Vec::new();
```

```
stack.push(1);
```

```
stack.push(2);
```

```
stack.push(3);
```



```
while let Some(top) = stack.pop() {  
    println!("{}", top);  
}
```

for 循环中使用模式来解构元组



■ for 可以获取一个模式。在 for 循环中，模式是 for 关键字直接跟随的值


```
let v = vec!['a', 'b', 'c'];
```

```
→ for (index, value) in v.iter().enumerate() {  
    println!("{}", value, index);  
}
```


函数参数



- 函数参数也可以是模式



```
fn foo(x: i32) {  
    // 代码  
}
```



```
fn print_coordinates(&(x, y): &(i32, i32)) {  
    println!("Current location: ({} , {})", x, y);  
}  
  
fn main() {  
    let point = (3, 5);  
    print_coordinates(&point);  
}
```

值 `&(3, 5)` 会匹配模式 `&(x, y)`，如此 `x` 得到了值 `3`，而 `y` 得到了值 `5`。

Refutability（可反驳性）：模式是否会匹配失效



- 模式有两种形式：refutable（可反驳的）和 irrefutable（不可反驳的）。
- 能匹配任何传递的可能值的模式被称为是 不可反驳的（*irrefutable*）。
 - 一个例子就是 `let x = 5;`
 - 因为 `x` 可以匹配任何值所以不可能失败。

Refutability（可反驳性）：模式是否会匹配失效



- 模式有两种形式：refutable（可反驳的）和 irrefutable（不可反驳的）。
- 能匹配任何传递的可能值的模式被称为是 不可反驳的（*irrefutable*）。
 - 一个例子就是 `let x = 5;`
 - 因为 `x` 可以匹配任何值所以不可能失败。
- 对某些可能的值进行匹配会失败的模式被称为是 可反驳的（*refutable*）
 - 一个这样的例子便是：
 - `if let Some(x) = a_value`

Refutability (可反驳性)：模式是否会匹配失效



- 通常我们无需担心可反驳和不可反驳模式的区别，不过确实需要熟悉可反驳性的概念，这样当在错误信息中看到时就知道如何应对。
- 以下代码能否通过编译？

```
let Some(x) = some_option_value;
```

Refutability (可反驳性)：模式是否会匹配失效



- 通常我们无需担心可反驳和不可反驳模式的区别，不过确实需要熟悉可反驳性的概念，这样当在错误信息中看到时就知道如何应对。

- 以下代码能否通过编译？

```
let Some(x) = some_option_value;
```

- 不能！若 `some_option_value` 的值是 `None`，则不会成功匹配模式 `Some(x)`，表明这个模式是可反驳的。

```
error[E0005]: refutable pattern in local binding: `None` not covered
-->
   |
3  | let Some(x) = some_option_value;
   |      ^^^^^^^ pattern `None` not covered
```


Refutability (可反驳性)：模式是否会匹配失效



- 通常我们无需担心可反驳和不可反驳模式的区别，不过确实需要熟悉可反驳性的概念，这样当在错误信息中看到时就知道如何应对。

- 以下代码能否通过编译？

```
let Some(x) = some_option_value;
```



```
if let Some(x) = some_option_value {  
    println!("{}", x);  
}
```

Refutability (可反驳性)：模式是否会匹配失效



- 通常我们无需担心可反驳和不可反驳模式的区别，不过确实需要熟悉可反驳性的概念，这样当在错误信息中看到时就知道如何应对。

- 以下代码能否通过编译？

```
if let x = 5 {  
    println!("{}", x);  
};
```

Refutability (可反驳性)：模式是否会匹配失效



- 通常我们无需担心可反驳和不可反驳模式的区别，不过确实需要熟悉可反驳性的概念，这样当在错误信息中看到时就知道如何应对。

- 以下代码能否通过编译？

```
if let x = 5 {  
    println!("{}", x);  
};
```

- 可以，但是Rust 会抱怨将不可反驳模式用于 if let 是没有意义的：

```
warning: irrefutable if-let pattern  
--> <anon>:2:5  
2 | /      if let x = 5 {  
3 | |      println!("{}", x);  
4 | |  };  
  | |  ^  
  |  _^  
= note: #[warn(irrefutable_let_patterns)] on by default
```

模式语法



- 匹配变量名。
- 请预测以下代码的运行结果：

```
fn main() {  
    let x = Some(5);  
    let y = 10;  
  
    match x {  
        Some(50) => println!("Got 50"),  
        Some(y) => println!("Matched, y = {:?}", y),  
        _ => println!("Default case, x = {:?}", x),  
    }  
  
    println!("at the end: x = {:?}", y = {:?}", x, y);  
}
```

模式语法



- 匹配变量名。
- 请预测以下代码的运行结果：


```
fn main() {  
    let x = Some(5);  
    let y = 10;  
  
    match x {  
        Some(50) => println!("Got 50"),  
        Some(y) => println!("Matched, y = {:?}", y),  
        _ => println!("Default case, x = {:?}", x),  
    }  
  
    println!("at the end: x = {:?}", y = {:?}", x, y);  
}
```

- 第二个匹配分支中的模式引入了一个新变量 `y`，它会匹配任何 `Some` 中的值。因为我们在 `match` 表达式的新作用域中，这是一个新变量，而不是开头声明为值 10 的那个 `y`。

模式语法



■ 多个模式:



```
let x = 1;

match x {
  1 | 2 => println!("one or two"),
  3 => println!("three"),
  _ => println!("anything"),
}
```

- 在 match 表达式中，可以使用 | 语法匹配多个模式，它代表 或 (or) 的意思

模式语法



- 通过 `..=` 匹配值的范围：

```
let x = 5;  
match x {  
  1..5 => println!("one through five"),  
  _ => println!("something else"),  
}
```

- 如果 `x` 是 1、2、3、4 或 5，第一个分支就会匹配。
- 相比使用 `|` 运算符表达相同的意思更为方便

模式语法



■ 解构分解值:

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let p = Point { x: 0, y: 7 };  
  
    let Point { x: a, y: b } = p;  
    assert_eq!(0, a);  
    assert_eq!(7, b);  
}
```

简化

```
struct Point {  
    x: i32,  
    y: i32,  
}  
  
fn main() {  
    let p = Point { x: 0, y: 7 };  
  
    let Point { x, y } = p;  
    assert_eq!(0, x);  
    assert_eq!(7, y);  
}
```


模式语法



- 解构分解值：
- 类似的，不只是Struct，Enum也可以解构
 - 通过let、match等模式匹配绑定内部的值：

模式语法

- 解构分解值:
- 类似的, 不只是Struct, Enum
- 通过let、match等模式


```
enum Message {  
    Quit,  
    Move { x: i32, y: i32 },  
    Write(String),  
    ChangeColor(i32, i32, i32),  
}
```

```
fn main() {  
    let msg = Message::ChangeColor(0, 160, 255);  
  
    match msg {  
        Message::Quit => {  
            println!("The Quit variant has no data to destructure.")  
        }  
        Message::Move { x, y } => {  
            println!(  
                "Move in the x direction {} and in the y direction {}",  
                x,  
                y  
            );  
        }  
        Message::Write(text) => println!("Text message: {}", text),  
        Message::ChangeColor(r, g, b) => {  
            println!(  
                "Change the color to red {}, green {}, and blue {}",  
                r,  
                g,  
                b  
            )  
        }  
    }  
}
```

模式语法



- 使用 `_` 忽略整个值:



```
fn foo(_: i32, y: i32) {  
    println!("This code only uses the y parameter: {}", y);  
}  
  
fn main() {  
    foo(3, 4);  
}
```

模式语法



- 使用 `_` 忽略整个值:

```
let mut setting_value = Some(5);
let new_setting_value = Some(10);

match (setting_value, new_setting_value) {
    → (Some(_), Some(_)) => {
        println!("Can't overwrite an existing customized value");
    }
    → _ => {
        setting_value = new_setting_value;
    }
}

println!("setting is {:?}", setting_value);
```

模式语法



- 使用 忽略整个值:

```
let numbers = (2, 4, 8, 16, 32);
```

```
match numbers {
```

```
    → (first, _, third, _, fifth) => {  
        println!("Some numbers: {}, {}, {}", first, third, fifth)  
    },  
}
```

这里忽略了一个五元元组中的第二和第四个值

模式语法



- 使用 `_` 忽略整个值：



```
fn main() {  
    let _x = 5;  
    let y = 10;  
}
```

通过在名字前以一个下划线开头来忽略未使用的变量；

这里得到了warning说未使用变量 `y`，不过没有warning说未使用下划线开头的变量。

模式语法



■ 使用 .. 忽略剩余值:

```
struct Point {  
    x: i32,  
    y: i32,  
    z: i32,  
}  
  
let origin = Point { x: 0, y: 0, z: 0 };  
  
match origin {  
    Point { x, .. } => println!("x is {}", x),  
}
```

这比不得不列出 `y: _` 和 `z: _` 要来得简单，特别是在处理有很多字段的结构体

模式语法



- 匹配守卫提供的额外条件：

```
let num = Some(4);

match num {
  → Some(x) if x < 5 => println!("less than five: {}", x),
    Some(x) => println!("{}", x),
    None => (),
}
```

匹配守卫 (*match guard*) 是一个指定于 `match` 分支模式之后的额外 `if` 条件，它也必须被满足才能选择此分支。


模式语法

■ *at* 运算符（@绑定）：

动机：

（1）我们希望测试 `Message::Hello` 的 `id` 字段是否位于 `3..=7` 范围内

（2）同时也希望能将其值绑定到 `id_variable` 变量中以便此分支关联的代码可以使用它

```
enum Message {  
    Hello { id: i32 },  
}  
  
let msg = Message::Hello { id: 5 };  
  
match msg {  
     Message::Hello { id: id_variable @ 3..=7 } => {  
        println!("Found an id in range: {}", id_variable)  
    },  
    Message::Hello { id: 10..=12 } => {  
        println!("Found an id in another range")  
    },  
    Message::Hello { id } => {  
        println!("Found some other id: {}", id)  
    },  
}
```



中山大學

SUN YAT-SEN UNIVERSITY

Q & A

Thanks!